

CSI 400 – Operating Systems

Information Regarding Programming Assignments

There will be 4-5 programming projects. Each will be due at 11:59PM (just before midnight) of the due date. Late submissions will be accepted but their score will be penalized by the the factor given by the following formula:

$$\text{Lateness Penalty Factor} = \text{MAX}(100\%, \frac{(\text{Number of seconds late})}{(\text{Number of seconds in a day})} * 25\%)$$

However, I might excuse the lateness penalty under extenuating circumstances if you communicate about them before the assignment is due. The due date and and any changes to the lateness penalty factor assignment will be indicated on the assignment sheet. They are really “projects”. That means they entail analysis of the problem parts, learning topics relevent to their solutions, planning for incremental development, design, coding, testing, debugging and the solution of unanticipated problems spread out over the assignment period. There will be pitfalls and problems to solve that you will not know about until you work through the project details! Most students who do no substantial work on a project before a couple of days before the due date will encounter frustration and failure.

1 Submission Information

For each program, you must electronically submit the required file(s). The procedure for the electronic submission is described later. (Please note that you should *not* mail the files to the professor or teaching assistant for grading. You might mail files to us for help on the project before it is due.)

2 Code Editing and Documentation

For each programming assignment, approximately 50% of the grade will be for correctness, 30% for the reports on experiments and explorations which are assigned for you to do with your programs and/or the system, the remaining 20% will be for structure and documentation.

However, the first thing we will do is test whether your Makefile(s) direct successful compilations and linkings (“builds”) of the programs we specify in the assignment. The builds will be tested under the following conditions:

1. The directories you submit will be reproduced.
2. All object and binary executable files will be removed.
3. The command “**gmake**” will be given with the working directory set to the directory containing the Makefile and the execution path set to the default of Albany’s `itsunix` systems.

If none of the required executable programs are successfully built, you will get 0 for the project, automatically! **It is your responsibility to compile, run and test your work on the itsunix system, and correct compilation errors, debug, and complete the project requirements you seek credit for before you submit it for grading.**

If your C++/C or assembly code is not consistantly indented to show syntactic structure, 5% will be deducted automatically. It is easy to get your C++/C code properly indented automatically when you use emacs. (Hint: C-C C-Q will re-indent the entire function that the cursor is currently in.)

The documentation comments to be written into each program must include:

- (a) You must give your name as the programmer at the beginning of each program file..
- (b) An overall description of what the program does must be given, however, it might simply refer **specifically** to a version or part name or number in the assignment. This is OK for assignments that specify the description in detail.

Any assumptions that you make with respect to the inputs (in addition to those stated in the assignment specification) must be clearly stated. In other words, if you can make your program give correct output under limited conditions, explain them in detail to maximize partial credit.

When you invent yourself what the program does, which may happen in later projects or more advanced courses, that must be written in terms of

1. what the inputs are, and
2. what the outputs are and how they depend on the inputs.

- (c) The purpose of every constant, data type and variable declared in your program must be stated at the point of declaration if a declaration is required. For assembly language programs this applies
- to memory variables, both static (allocated in the `.data` segment) and automatic (allocated in a stack frame) AND
 - to registers. Document the purpose of every register you use.
- (d) For each function you must provide a description of what the function does *in terms of* a description of all the parameters, what the values returned (as return value or through reference parameters) are, and how they and any other effects depend on the parameters.
- (e) You must also include in-line comments (i.e., comments interspersed with code) to explain any logic that is not obviously expressed by the code itself, such as loop invariants and explanations of how program operation depends on assumptions about the inputs.

Document key uses of variables with **invariant comments** about relationships among data that are *clearly stated and absolutely true* whenever control passes the invariant comment. For example, instead of the rather uninformative comments in

```
lbu  $s3, str($s2)  #Use string index in $s2 to load a byte into $s3
addi $s2,$s2,1     #Increment counter
```

instead write

```
lbu  $s3, str($s2)  # $s3 contains the INPUT char to be processed now.
addi $s2,$s2,1     # $s2 is the number of chars that had been loaded.
```

The better choice for `$s3` reminds us that the input string is in memory beginning at address `str` and tells us the significance of the current value in `$s3`. Everybody already knows `lbu` loads a byte. The comment for the `addi` instruction documents the details of how `$s2` is managed. Everybody already knows `addi` increments something!

A useful feature of an invariant comment is it can sometimes be expressed as a test you can program into your code to test itself for internal consistency (also known as “sanity checks”). The C/C++ standard `assert` macro (whose definition comes from `#include <assert.h>`) evaluates a boolean expression and makes the program print the assertion if it is false (“fires”) and then crash. The resulting “core dump file” can be analyzed with a debugger. If the program is run under debugger control, it will stop at the fired assertion.

3 Modularization Rules

Some of the projects will explicitly require that the program be designed and implemented with separately compiled modules whose **interfaces are defined in header files**. Each separate module should

then be separately tested with a simple driver program you write specifically to test that one module. The more complicated code that will use this module is not tried until this module is completely debugged **by itself**, with the help of the test driver.

Putting your development and testing activities into such a disciplined sequence is an example of a “process requirement.” The requirement to write documentation for each module’s interface into the module’s header file is another example of a process requirement. Specified process requirements do count for grading purposes. (Note the word “process” in this paragraph means something different from its meaning in the operating system course’s lecture and textbook material.)

The fundamental idea for **logical organization** of software is to organize it into **modules**: Each module solves a separate problem and the modules cooperate by using each other’s services or data. Modularization helps make the creation of highly complex software feasible. Now that you know from the prerequisite courses CSI201 and CSI310 some of the computer problems and algorithms that are feasible to handle in small programs, this course will introduce practices that make feasible more ambitious, realistically sized projects.

One of these practices is to distribute the code over several different files. Major practical programs may be written in many thousands of files! The distribution of the code into different files (or directories, libraries or networked systems too) is called the **physical organization**. This practice is most beneficial when:

The physical organization reflects the logical organization.

Different programming languages and environments (for example, C, C++, Java, and assembly languages; and separated tool Unix-like, integrated, and computer aided software design tools) provide different styles and levels of support for modularization expressed through physical organization. This course will cover the support provided by traditional C/C++ header, implementation and separately compiled object files with the C/C++ preprocessor, plus some object orientation.

4 Review Questions

1. What is the maximum number of points you will get for a project submission that we cannot compile and run?
2. What is a process rule?
3. Give some reasons why it’s not a good idea to start a programming project 2 days or less before it is due.
4. The following question was posed for subjects in which I required that test cases for project programs be developed, used and submitted. For CSI400 this is not required for all projects. So, for enrichment purposes, please find out and answer, possibly by a Web search, “What is regression testing? Why do you think it is not done as much as it should be in some organizations?”
5. What is special about an “invariant comment”?
6. If you read an invariant comment aloud, would it always be a complete sentence? Can it be a question?
7. Programming textbooks and lecture material often have code examples with comments to explain something about the programming language feature that is being illustrated. Are comments of this nature required in this course? Think of some reasons to write them into you project or lab exercise code, or into a production project anyway. Think of some reasons to omit them.

8. Experiment with the `assert` macro in a simple C++ program. Remember to “`#include <assert.h>`” in the source file. Get the assertion to “fire” while running the program under the `gdb` or `ddd` debugger.
9. Generally speaking, what should go into a header file? Base your answer on what is written above.
10. Discuss these terms, preferably in your own words: module, logical organization, physical organization.
11. Where are static variables allocated? Where are automatic variables allocated? Why is a stack sometimes called a LIFO store? In which course did you learn about stacks? Implement a stack in C++.